Effective Keyword Search in Relational Databases

Fang Liu, Clement Yu
Computer Science Department
University of Illinois at Chicago
{fliu1,yu}@cs.uic.edu

Weiyi Meng
Computer Science Department
Binghamton University
meng@cs.binghamton.edu

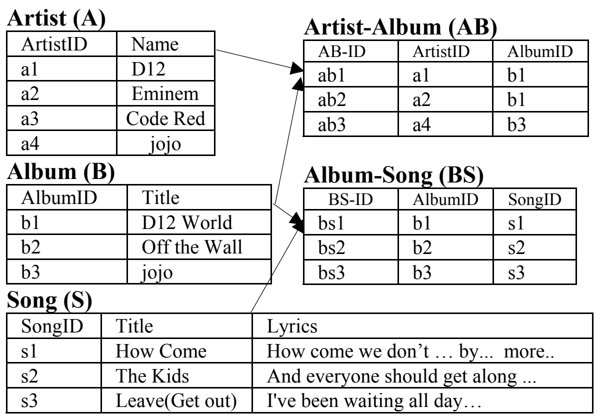
Abdur Chowdhury
Search & Navigation Group
America Online, Inc.
cabdur@aol.com

ABSTRACT

With the amount of available text data in relational databases growing rapidly, the need for ordinary users to search such information is dramatically increasing. Even though the major RDBMSs have provided full-text search capabilities, they still require users to have knowledge of the database schemas and use a structured query language to search information. This search model is complicated for most ordinary users. Inspired by the big success of information retrieval (IR) style keyword search on the web, keyword search in relational databases has recently emerged as a new research topic. The differences between text databases and relational databases result in three new challenges: (1) Answers needed by users are not limited to individual tuples, but results assembled from joining tuples from multiple tables are used to form answers in the form of tuple trees. (2) A single score for each answer (i.e. a tuple tree) is needed to estimate its relevance to a given query. These scores are used to rank the most relevant answers as high as possible. (3) Relational databases have much richer structures than text databases. Existing IR strategies are inadequate in ranking relational outputs. In this paper, we propose a novel IR ranking strategy for effective keyword search. We are the first that conducts comprehensive experiments on search effectiveness using a real world database and a set of keyword queries collected by a major search company. Experimental results show that our strategy is significantly better than existing strategies. Our approach can be used both at the application level and be incorporated into a RDBMS to support keyword-based search in relational databases.

1. INTRODUCTION

The amount of available structured data (in internet or intranet or even on personal desktops) for ordinary users grows rapidly. Besides data types such as number, date and time, structured databases usually also contain a large amount of text data, such as names of people, organizations and products, titles of books, songs and movies, street addresses, descriptions or reviews of products, contents of papers, and lyrics of songs, etc. The need for ordinary users to find information from text in these databases is dramatically increasing. The objective of this paper is to provide effective search of text information in relational databases. We take a lyrics database (Figure 1) as an example to illustrate the problem. There are five tables in the lyrics database. Table *Artist* has one text column: *Name*. Table *Album* has one text column: *Title*. Table *Song* has two text columns: *Title* and *Lyrics*. The tuples of Table *Artist* and those of Table *Album* have *m:n* relationships (an album may be produced by multiple artists and an artist may produce more than one album), and Table *Aritst-Album* is the corresponding relationship table. Table *Song-Album* is also a relationship table capturing the *m:n* relationships between tuples of *Album* and *Song* (a song may be contained in multiple albums and an album many contain more than one song). Note that Table *Aritst-Album* and Table *Aritst-Album* do not have other columns except their primary keys and foreign keys.



Artist (A)

ArtistID	Name
a1	D12
a2	Eminem
a3	Code Red
a4	jojo

Artist-Album (AB)

AB-ID	ArtistID	AlbumID
ab1	a1	b1
ab2	a2	b1
ab3	a4	b3

Album (B)

AlbumID	Title
b1	D12 World
b2	Off the Wall
b3	jojo

Album-Song (BS)

BS-ID	AlbumID	SongID
bs1	b1	s1
bs2	b2	s2
bs3	b3	s3

Song (S)

SongID	Title	Lyrics
s1	How Come	How come we don't … by... more..
s2	The Kids	And everyone should get along ...
s3	Leave(Get out)	I've been waiting all day…

Figure 1: Lyrics Database Example

The traditional search model in relational databases requires users to have knowledge of the database schema and to use a structured query language such as SQL or QBE-based interfaces. Even though most of the major RDBMSs have integrated full-text search capabilities using relevance-based ranking strategies developed in information retrieval (IR), they still have the above two requirements for users. Suppose a user is looking for albums titled "off the wall" and he/she cannot remember the exact title. A typical SQL query is shown in Figure 2 and the tuple *b2* is expected to be one of the top ranked results. Obviously, this model of search is too complicated for ordinary users.

```
Select * from Album B
Where Contains (B.title, 'off wall', 1) > 0
Order by score(1) desc
```

Figure 2: An Oracle SQL Example

Query 1:	*"off wall"*
Query 2:	*"lyrics how come by D12"*
Query 3:	*"album by D12 and Eminem"*

Tuple Tree 1: $b2$

Tuple Tree 2: $a1 \rightarrow ab1 \leftarrow b1 \rightarrow bs1 \leftarrow s1$

Tuple Tree 3: $a1 \rightarrow ab1 \leftarrow b1 \rightarrow ab2 \leftarrow a2$

Figure 3: Queries and Tuple Trees

With the tremendous success of web search engines, keyword search has become the most popular search model for ordinary users. Users do not need to know the database schema or use a structured query language. Instead, they submit a list of keywords using a very simple interface, and a search engine returns ranked documents based on relevance to the query from its text databases. Therefore, it is extremely desirable to support the keyword search model in relational databases. With such support, a user can avoid writing a SQL query; and he/she can just submit a simple keyword query "off wall" to the lyrics database.

Applying the keyword search techniques in text databases (IR) to relational databases (DB) is a challenging task because the two types of databases are different. First, in text databases, the basic information units searched by users are documents. For a given keyword query, IR systems compute a numeric score for each document and rank the documents by this score. The top ranked documents are returned as answers. In relational databases, however, information is stored in the form of columns, tables and primary key to foreign key relationships. The logical unit of answers needed by users is not limited to an individual column value or even an individual tuple; it may be multiple tuples joined together. Consider Query 1 as shown in Figure 3. Tuple Tree 1 is one desired answer (though the query and the text value are not exactly matched). Note that, in Figure 3, an edge from *a1* to *ab1* denotes a primary key to foreign key relationship. For Query 2, one desired answer is a joining tree of five tuples (Tuple Tree 2 as shown in Figure 3). Note that the query matches both text columns (*title* and *lyrics*) values in the tuple *s1*. For Query 3, one desired answer is a joining tree of five tuples (Tuple Tree 3). Second, effectiveness is a key factor for the success of keyword search. Even in a medium sized database, there may be dozens of candidate answers for an ordinary keyword query. Consider the query "off wall" again. There exists many titles of songs, titles of albums or lyrics of songs that contain both keywords "off" and "wall". Therefore, there are many tuple trees that can be answers for the query. However, these answers are not equally useful to the user. We need to rank the more relevant answers higher. Otherwise, users will be discouraged to use keyword search. Note that a key reason behind the tremendous success of web search engines is that their ranking strategies are highly effective that, for many queries, they can rank relevant web pages in the first ten results out of billions of web pages. Due to the difference in the basic answer unit between document searches and database searches, in relational databases, we need to assign a single ranking score for each tuple tree, which may consist of multiple tuples with text columns, in order to rank the answers effectively. The characteristics of text columns are usually diverse. For example, some text columns such as people's names and album titles are very short, while other text columns such as song lyrics are much longer. In contrast, in text databases, we only need to compute a score for each single document. Thus, the ranking strategy for relational databases needs to consider more factors and is more complicated. Third, relational databases have much richer structures than text databases. For a given query, we desire answers to be returned with semantics. For example, for Query 2 and one answer Tuple Tree 2, it is quite obvious that the sub-query "how come" is the *Song* title of the tuple *s1*, and the sub-query "D12" is the *Artist* name of the tuple *a1*, although all individual keywords (except "D12") in the two queries are very common words, and some individual keywords also appear in other text values of Tuple Tree 2 (e.g., "D12" appears in the title of *b1*, and "how" and "come" appear in lyrics of *s1*). We consider the correspondences between sub-queries in a given query and database columns in an answer as the semantics of the query in the answer. In summary, in relational databases, we have three key steps for processing a given keyword query. (1) Generate all candidate answers, each of which is a tuple tree by joining tuples from multiple tables. (2) Then compute a single score for each answer. The scores should be defined in such a way so that the most relevant answers are ranked as high as possible. (3) And finally return answers with semantics.

Recently, keyword search on relational databases has merged. DBXplorer [1], DISCOVER [10], BANKS [2], and Hristidis et al. [11] are systems that support keyword search on relational databases. For the first step, they generate tuple trees from multiple tables as answers. The first three systems require an answer containing all keywords in a query, while the last one only requires an answer containing some but not necessarily all keywords in the query. Efficiency has been the focus for the first step [1, 2 10, 11]: rules are designed to avoid generation of unnecessary tuple trees, and more efficient algorithms are proposed to improve the time and space complexities. For the second step, the first two systems use a very simple ranking strategy: the answers are ranked in ascending order of the number of joins involved in the tuple trees. When two tuple trees have the same number of joins, their ranks are determined arbitrarily. Thus, all tuple trees consisting of a single tuple are ranked ahead of all tuples trees with joins. The ranking strategy of the BANKS system is to combine two types of information in a tuple tree to compute a score for ranking: a weight (similar to PageRank for web pages) of each tuple, and a weight of each edge in the tuple tree that measures how related the two tuples are. The strategy of DBXplorer [1] and DISCOVER [10] and the strategy of BANKS [2] for the second step do not utilize any state-of-the-art IR ranking methods, which have been tremendously successful. In a database that contains a large amount of text data, these strategies will be shown not to work well. Hristidis et al. [11] propose a strategy by applying IR-style ranking methods into the computation of ranking scores in a straightforward manner. They consider each text column as a collection and each value in the text column as a document. A state-of-the-art IR ranking method is used to compute a score between a given query and each text column value in the tuple tree. A final score is obtained by dividing the sum of all these scores by the number of tuples (i.e. the number of joins plus 1) in the tree. However, they only concentrate on the efficiency issue of the implementation of the ranking strategy and do not conduct any experiments on the effectiveness issue. In addition, our experimental results show that their strategy ignores some important factors that are critical for search effectiveness.

Our paper focuses on search effectiveness. We propose a novel ranking strategy that ranks answers (tuple trees) effectively and returns answers with basic semantics. This strategy can be used both at the application level and be incorporated into a RDBMS to support keyword search in relational databases. We adapt the framework proposed in Hristidis et al. [11] to generate tuple trees as answers for a given query. Efficiency issues are not investigated in this paper. We emphasize that effectiveness of keyword search of text data (in both text and structured databases) is at least as important as efficiency.

Our key contributions are as follows:

1. We identify four new factors that are critical to the problem of search effectiveness in relational databases.
2. We propose a novel ranking strategy to solve the effectiveness problem. Answers are returned with basic semantics.
3. We are the first to conduct comprehensive experiments for the effectiveness problem.
4. Experimental results show that our strategy is significantly better than existing works in effectiveness (77.4% better than [11] and 16.3% better than Google[1]).

In Section 2, we briefly introduce the framework for generation of answers (tuple trees) for keyword search in relational databases. In Section 3, we briefly introduce the background of a recent IR similarity function. In Section 4, we analyze new challenges for keyword search in relational databases, identify new factors that are critical to ranking effectiveness, and propose a novel ranking strategy. In Section 5, we present experimental results to demonstrate the effectiveness of our ranking strategy. Section 6 discusses related works. Section 7 draws the conclusion and outlines directions for future work.

2. ANSWER GENERATION

In this section, we describe the framework for generating answers for given keyword queries (a modification of [11]). Section 2.1 describes what an answer is for a keyword query in relational databases. Section 2.2 gives an algorithm to generate answers.

2.1 Tuple Trees As Answers

We use a graph to model a database schema. A **schema graph** is a directed graph *SG*. For each table R_i in the database, there is a node in the schema graph. If there is a primary key to foreign key relationship from the table R_i to the table R_j in the database, then there is an edge from the node R_i to the node R_j in the schema graph. Each table R_i has m_i (m_i>=0) text columns $\{c^i_1, c^i_2, \ldots, c^i_{mi}\}$. Figure 4 shows a schema graph of the lyrics database (columns are shown in Figure 1). A **tuple tree** *T* is a joining tree of tuples. Each node t_i in *T* is a tuple in the database. Suppose (R_i, R_j) is an edge in the schema graph. Let $t_i \in R_i$, $t_j \in R_j$, and (t_i *join* t_j) $\in$ (R_i *join* R_j). Then (t_i, t_j) is an edge in the tuple tree *T*. The **size** of a tuple tree *T* is the number of tuples involved. Note that a single tuple is the simplest tuple tree with size 1. A **keyword query** *Q* consists of a list of keywords $\{k_1, k_2, \ldots k_n\}$. For a given query *Q*, an **answer** must be a tuple tree *T* that satisfies the following conditions: (1) every leaf node t_i in *T* contains at least one keyword in *Q* (more precisely, at least a text column value of the tuple t_i contains at least one keyword in *Q* and different leaf nodes may contain the same keyword), and (2) each tuple only appears at most once in the tuple tree. Note that a non-leaf node (a node has two or more edges) may or may not contain any keyword. This definition implies that (a) if an answer contains multiple tuples, they must be joined together as a tree, (b) we assume the OR semantics (i.e. an answer must contain at least one query keyword) for answering a query, and (c) there is no redundancy, because if we remove any leaf node, we will lose the corresponding keyword match between the query and the node, and if we remove any non-leaf node, the tuple tree becomes

[1] Although Google retrieves from text databases (not from relational databases), its big success necessitates a comparison.

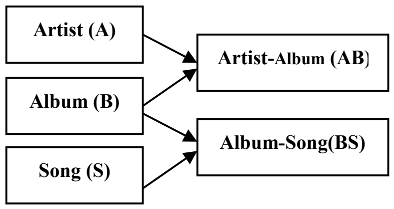


Figure 4: The Lyrics Database Schema Graph

$a1 \rightarrow ab1 \leftarrow b1 \rightarrow bs1 \leftarrow s1$ **(Tuple Tree 2)**

$a1 \rightarrow ab1 \leftarrow b1 \rightarrow ab2 \leftarrow a2$ **(Tuple Tree 3)**

$A^{Q} \rightarrow AB^{F} \leftarrow B^{F} \rightarrow BS^{F} \leftarrow S^{Q}$ **(Answer Graph 2)**

A^Q join AB^F on (A^Q.ArtistID = AB^F.ArtistID) join B^F on (AB^F.AlbumID =B^F.AlbumID) join BS^F on (B^F.AlbumID = BS^F .AlbumID) join S^Q on (BS^F.SongID = S^Q. SongID)

$A^{1,Q} \rightarrow AB^{1,F} \leftarrow B^{F} \rightarrow AB^{2,F} \leftarrow A^{2,Q}$ **(Answer Graph 3)**

…join.. Where ($A^{1,Q}$.ArtistID != $A^{2,Q}$.ArtistID) and ($AB^{1,F}$ $AB^{1,F}$ AB-ID!= $AB^{1,F}$.AB-ID)

Figure 5: Tuple Trees, Answer Graphs, Join Expressions

disconnected. Take Tuple Tree 2 and Tuple Tree 3 in Section 1 as examples (also shown in Figure 5). In Tuple Tree 2, the leaf nodes *a1* contains the keyword "D12", and *s1* contains two keywords "how" and "come" in Query 2. In Tuple Tree 3, leaf nodes *a1* and *a2* contain the keywords "D12" and "Eminem" in Query 3 respectively; the non-leaf node *b1* also contains the keyword "D12". Note that we consider a tuple tree as an answer as long as it satisfies the definition. An answer may or may not be relevant to a given query.

2.2 Answer Graph to Generate Answers

In this section, we briefly describe an algorithm (a modification of the algorithm described in [11]) to generate tuple trees as answers. For a given keyword query *Q*, the **query tuple set** R^Q of a table *R* is defined as the set of all tuples in *R* that contain at least one keyword in *Q*. For example, the query tuple sets of Table *Artist* for Query 1, Query 2 and Query 3 are A^{Q1}={}, A^{Q2}={*a1*} and A^{Q3}={*a1, a2*} respectively. We define the **free tuple set** R^F of a table *R* as the set of all tuples in *R*. For example, the free query set for Table *Artist* is A^F={*a1, a2, a3*}. Implied from the definition of an answer, for a given query *Q*, if a tuple tree *T* is an answer, then each leaf node t_i in Table R_i belongs to the query tuple set R_i^Q, and each non-leaf node t_j in Table R_j belongs to the free tuple set R_j^F. We use R^{QorF} to denote a **tuple set**, which may be either a query tuple set or a free tuple set. Due to the existence of *m:n* relationships (for example, an album may be produced by more than one artist), tuple sets of the same table may appear more than once in a join expression. If this happens, each occurrence of the same tuple set is considered to be a different **alias of the tuple set**. Now, we define an answer graph as a join expression on alias of tuple sets that produces tuple trees as answers. An **answer graph** is a directed graph *AG* of alias of tuple sets, where each leaf node $R_i^{x,Q}$ is the *x-th* alias of the query tuple set R_i^Q of the table R_i and each non-leaf node $R_j^{y,F}$ is the *y-th* alias of the free tuple set R_j^F of the table R_j, and each edge ($R_i^{x,QorF}$,$R_j^{y,QorF}$) corresponds to an edge (R_i, R_j) in the schema graph *SG* and this edge also corresponds to a join clause "$R_i^{x,QorF}$ join $R_j^{y,QorF}$ on $R_i^{x,QorF}$.PrimaryKey= $R_j^{y,QorF}$.ForeignKey". We define the size of an answer graph as the number of nodes. Obviously, the size of an answer graph is the same as that of a tuple tree it produces. For example, the Answer Graph 2 and Answer Graph 3 (as well as the

```
Input: query $Q$, schema graph $SG$, $maxn$, $MAXN$
Output: a set of answer graphs $S$
1. Generate all non-empty query tuple sets {$R_1^Q$, $R_2^Q$,...$R_n^Q$ } and all non-empty free tuple sets {$R_1^F$, $R_2^F$,...$R_n^F$ } for $Q$. Add the two sets of tuple sets into a queue $E$. Note that, in the following steps, these tuple sets are considered as symbols.
2. While $E$ is not empty {
    Pop the head $h$ from $E$.
    If $h$ violates the constraints of $MAXN$ and $maxn$, then $h$ is discarded
    Else {
        If $h$ is a valid answer graph+, then add it to $S$.
        For each $R_i^{\text{Q or F}}$ in $h$
            For each $R_j$ that is adjacent to $R_i$ in SG {
                Add $R_j^Q$ with an edge into $h$ to get a new $h1$.
                Add $R_j^F$ with an edge into $h$ to get a new $h2$.
                If $h1$ is not in $E$, then push it into the end of $E$.
                If $h2$ is not in $E$, then push it into the end of $E$.
            }
    }
}
3. For each graph of tuple sets in $S$, if there is more than one occurrence for any tuple set, give these occurrences different alias names.
4. Return $S$.

+Note that whether an $h$ is a valid answer graph is determined by the definition of an answer graph, as well as the two parameters $maxn$ and $MAXN$.
```

Figure 6: Algorithm for Answer Graph Generation

corresponding join expressions) that generate Tuple Tree 2 and Tuple Tree 3 are shown in Figure 5 (If there is only one occurrence of a tuple set, we omit the alias number). In Answer Graph 3, there are two alias of tuple sets $A^{1,Q}$ and $A^{2,Q}$ for the same query tuple set A^Q of Table A. If there exists more than one alias for a tuple set, in the corresponding join expression, we need to add a condition such as "$A^{1,Q}$.ArtistID != $A^{2,Q}$.ArtistID" to avoid producing tuple trees such as $a1 \rightarrow ab1 \leftarrow b1 \rightarrow ab1 \leftarrow a1$, which violates the second condition of the definition of an answer. If there is an edge (R_i, R_j) in the schema graph SG, and n is the maximum number of distinct tuples in the foreign table R_j that can be joined with one tuple in the primary table R_i, then, in theory, each tuple set of R_i may connect to n tuple sets of R_j in an answer graph in order to produce all possible tuples trees, each of which contains one tuple of R_i that is joined with at most n distinct tuples of R_j. And the number of answer graphs is only data bounded by the query and the database. Thus, we set up two parameters, *maxn* (for each tuple set of a primary table, it is the maximum number of tuple sets of a foreign table that can be joined) and *MAXN* (the maximum number of tuple sets in an answer graph) to avoid generating complicated but less meaningful answer graphs. Note that, for a given query, there is usually more than one answer graph. For example, besides Answer Graph 2, A^{Q2} and S^{Q2} (S^{Q2}={s1}) are also two answer graphs for Query 2. Figure 6 gives a breadth-first search algorithm to generate all answer graphs *AGs* for a given query Q and a given schema graph *SG*. With answer graphs, it is rather straightforward to produce tuple trees as answers by evaluating the corresponding join expressions. It can be proved that the answer graphs output by our algorithm can produce all and only all answers (tuple trees) if we do not apply the constraints of *maxn* and *MAXN*.

In summary, in order to generate answers for a given query, the system first finds all tuple sets, then it generates all answer graphs using the tuple sets and the schema graph, and finally, it produces all tuple trees as answers by evaluating the join expressions in the answer graphs. The efficiency issue of answer generation is not the focus of this paper and will not be discussed.

3. BACKGROUND IN IR RANKING

Modern IR systems rank documents for a given query. The higher a document is ranked, the more likely the document is relevant or useful to the query. In Section 3.1 we describe how the effectiveness of IR ranking is evaluated. The IR effectiveness measures will be used in our experiments. In Section 3.2, we introduce and discuss a recent IR similarity function and analyze some critical factors that affect search effectiveness.

3.1 Effectiveness Measures in IR

In IR, there are many measures to evaluate effectiveness. **11-point precision and recall** (precision is the number of relevant documents retrieved divided by the number of retrieved documents, and recall is the number of relevant documents retrieved divided by the number of relevant documents) is a standard measure. At each of the 11 recall levels (0, 0.1, 0.2,…,1), a precision value is computed. These 11 precisions are usually plotted in a graph to illustrate the overall effectiveness as well as the trade off between precision and recall. Mean average precision (**MAP**) is another standard measure. A precision is computed after each relevant document is retrieved. Then we average all precision values to get a single number to measure the overall effectiveness.

For ranking tasks in which users look for a single or a very small set of target documents (such as homepage search and question answering [19]) in a large collection, the **reciprocal rank** is another popular measure. For a given query, the reciprocal rank is 1 divided by the rank at which the first correct answer is returned or 0 if no correct answer is returned. For example, if the first relevant document is ranked at 5, then the reciprocal rank is 1/5.

In IR experimentation, values of each of the above measures are usually averaged over a set of queries to get a single number to evaluate the effectiveness of an IR system. Note that "*evaluation of search effectiveness has been a cornerstone of IR*" [18], and effectiveness is one of the two (efficiency is the other one) most important problems in IR. This paper is the first work that conducts a comprehensive effectiveness evaluation on the problem of keyword search in relational databases.

3.2 Ranking Model in IR

To rank documents, IR systems assign a score for each document as an estimation of the document relevance to the given query. The widely used model to compute such a score is the vector space model [26]. Each text (both documents and queries) is represented as a vector of terms, each of which may be an individual keyword or a multi-word phrase. The vocabulary of terms makes up a term space. Each term occupies a dimension in the space. Each text (a document or a query) is represented as a vector on this term space, and each item in the vector of a text has a non-negative weight, which measures the importance of the corresponding term k in the text. Thus, a similarity value between a document vector D and a query vector Q can be computed as the ranking score.

Formula 1 shows the inner product (dot product) function to compute the similarity. Weighting a term in a document (the *weight(k, D)* component in Formula 1) is the most critical problem in computing similarity values. Formula 2 shows the

$$Sim(Q,D) = \sum_{k \in Q,D} weight(k,Q) * weight(k,D) \quad \textbf{(1)}$$

$$weight(k,D) = \frac{ntf}{ndl} * idf \quad \textbf{(2)}$$

$$ntf = 1 + \ln(1 + \ln(tf)) \quad \textbf{(2.1)}$$

$$idf = \ln \frac{N}{df + 1} \quad \textbf{(2.2)}$$

$$ndl = (1 - s) + s * \frac{dl}{avgdl} \quad \textbf{(2.3)}$$

pivoted normalization weighting method [17, 18], which is one of the most widely used weighting methods in IR. Note that, conceptually, we put the *idf* component into Formula 2 but not into Formula 1. The weight of a term in a document is determined by the following three factors.

- Term Frequency (*tf* in Formula 2.1): the number of occurrences of a term in a document. Intuitively, the more a term occurs in a document, the higher the weight of the term should be. However, the same term may occur many times in a long document, and the importance of a term should not be linearly dependent on the raw *tf* when *tf* is rather large. It has been accepted in IR that the raw *tf* should be dampened. Formula 2.1 applies the log function twice to normalize the raw *tf* to get *ntf.*
- Document Frequency (*df* in Formula 2.2): the number of documents that a term occurs in a collection. By intuition, in a collection, the more documents a term appears in, the worse discriminator it is, and it should be assigned a smaller weight. Formula 2.2 shows the inverse document frequency (*idf*) weighting method to normalize *df*: dividing the total number of documents (*N* in Formula 2.2) by (*df*+1) and then applying the *log* function
- Document Length (*dl* in Formula 2.3): the length of a document in bytes or in number of terms contained in the document. Because longer documents contain more terms and higher term frequencies, longer documents tend to have higher inner product values for a given query. Formula 2.3 provides a normalization to reduce the term weights in long documents, where *avgdl* is the average document length in the collection, and *s* is a constant and is usually set to 0.2.

Weighting a term in a query (the *weight(k, Q)* component in Formula 1) is rather simple: We use raw term frequency (*qtf*) in the query. Note that normalization on the above three factors has significantly improved search effectiveness in IR than the simple *tf*idf* weighting methods [18].

4. NOVEL RANKING STRATEGY FOR RELATIONAL DATABASES

In this section, we propose a novel ranking strategy for effective keyword search in relational databases. In IR, a document is a basic information unit stored in a text database; and it is also the basic unit of answers needed by users. A similarity value between a given query and a document is computed to rank documents. However, the basic text information unit stored in a relational database is a text column value, while the basic unit of answers needed by users is a tuple tree, which is assembled by joining multiple tuples, each of which may contain zero, one or multiple text column values (each text column value is considered as a document). A similarity value between a given query and a tuple

$$Sim(Q,T) = \sum_{k \in Q,T} weight(k,Q) * weight(k,T) \quad \textbf{(3)}$$

tree needs to be computed to rank tuple trees. This value has two factors: similarity contributed from each text column value in the tuple tree, and a combination of all these contributions. Let *T* be a tuple tree and $\{D_1, D_2, ..., D_m\}$ be all text column values in *T*. We define each text column value D_i as a **document** and *T* as a **super-document.** Then we can compute a similarity value between the query *Q* and the super-document *T* as shown in Formula 3 to rank tuple trees. The similarity is the dot product of the query vector and the super-document vector. The method of weighting a term in the query, *weight(k,Q)*, still uses the term's raw *qtf* (term frequency in the query). Our focus is on ***weight(k,T)***, the weight of a term *k* in a super-document *T.*

In Section 4.1, we introduce the ranking strategy proposed by Hristidis et al. [11]. In Section 4.2, we identify four important factors that affect search effectiveness and propose a novel term weighting strategy. Section 4.3 identifies the schema term problem and proposes a solution. Section 4.4 proposes phrase-based and concept-based search models that improve effectiveness and can return semantics.

4.1 Ranking Strategy in Related Work

The weighting method in Hristidis et al. [11] considers each text column as a **collection**, and uses the standard IR weighting method as shown in Formula 2 to compute a weight for each term *k* in each document D_i. Then, as shown in Formula 4, each weight

$$weight(k,T) = \sum_{D_i \in T} weight(k,D_i) / size(T) \quad \textbf{(4)}$$

is normalized (divided by *size(T)*, i.e. the number of tuples in *T*). The weights of the term in all documents are summed to obtain the term weight in the super-document *T.* Formula 4 identifies and deals with a new factor, *size(T)*, that affects similarity. However, more factors need to be considered.

4.2 Four Normalizations

We identify four important factors that affect search effectiveness and propose a novel term weighting strategy as shown by Formulas 5.1 and 5.2. Formula 5.1 computes a term's weight in a document D_i, and Formula 5.2 computes the same term's weight

$$weight(k,D_i) = \frac{ntf * idf^g}{ndl * Nsize(T)} \quad \textbf{(5.1)}$$

$$weight(k,T) = Comb(weight(k,D_1),...,weight(k,D_m)) \quad \textbf{(5.2)}$$

in the tuple tree *T*. *ntf* is still computed using Formula 2.1; *Nsize(T)* is a new tuple tree normalization factor (see Section 4.2.1 and Formula 6); *ndl* is a new document length normalization factor (Section 4.2.2 and Formula 7); idf^g is a new inverted document frequency weight (Section 4.2.3 and Formula 8); *Comb()* is a new function to combine term weights in documents into a term weight in a tuple tree (Section 4.2.4 and Formula 9).

4.2.1 Tuple Tree Size Normalization

The tuple tree size factor, *size(T)*, is similar to the document length (*dl*) factor discussed in Section 3.2 in the following sense: a tuple tree with more tuples tends to contain more terms and higher term frequencies. However, using the raw *size(T)* as shown in Formula 4 can be sub-optimal, especially for a complex query whose relevant answers are tuple trees involving multiple tuples, each of which contains a subset of the query keywords. Consider the 26th query Q ("jojo leave lyrics") in the appendix and the example in Figure 1. A relevant answer is a tuple tree $b3 \rightarrow bs3 \leftarrow s3$. Let us call it T_3. Let $T4$ be the single-node tuple tree that only contains the tuple *b3*, which contains the keyword "jojo", and $T5$ be the single-node tuple tree that only contains the tuple *s3*, which contains the keyword "leave". Both T_4 and T_5 are answers for the query under OR semantics, but they are not as relevant as T_3 to the query. Suppose *weight("jojo",b3) = w1* and *weight("leave",s3) = w2* (all other weights are zeros). Then $Sim(Q,T_4) = w1$, $Sim(Q,T_5) = w2$ and $Sim(Q,T_3) = (w1+w2)/3$. Obviously, at least one of T_4 and T_5 will be ranked ahead of T_3 (if we ignore the tie situation), and this ranking is ineffective because T_3 should be ranked higher than both T_4 and T_5. Thus, *size(T)* should be normalized. We borrow the Formula 2.3 for document length normalization to tuple tree size normalization: the weight of a term in a document is divided by *Nsize(T)* as shown in Formula 5.1 and Formula 6 instead of the raw *size(T)*. For

$$Nsize(T) = (1-s) + s * \frac{size(T)}{avgsize} \qquad \textbf{(6)}$$

document length normalization, the *avgdl* is averaged on all documents in the collection. Thus, one way to compute *avgsize* is to generate all possible answer graphs using the algorithm described in Figure 6 by using a pseudo query Q_p and assuming all query tuple sets and all free tuple sets are non-empty, and then average the *sizes* of all the answer graphs With this normalization, the new similarity values for T_3, T_4 and T_5 become *w1, w2* and *(w1+w2)/1.15* respectively. As a result, using *Nsize(T)*, T_5 is much more likely to be ranked ahead of both T_3 and T_4 than using the raw *size(T)*. Note that this normalization is independent of any keyword in the query and any document in the tuple tree.

4.2.2 Document Length Normalization Reconsidered

When we logically combine multiple documents into one super-document *T*, the document length factor needs to be reconsidered because collections (i.e. text columns) have their own *avgdl* values, which may be very different. Take Query 2 as an example. In the answer Tuple Tree 2, the keywords "how" and "come" occur in both D_{Title} (i.e. the *Title* column value in the tuple *s1*) and D_{Lyrics} (i.e. the *Lyrics* column value in the tuple *s1*). The original document length normalization is within each collection (we call each such collection a **local collection** which corresponds to a text column). If both the length of D_{Title} and the length of D_{Lyrics} are equal to the *avgdl* values (3.21 and 239 as shown in Table 1) in *Title* and *Lyrics* respectively, then the *ndl* values computed by Formula 2.3 are 1 for both documents, while D_{Lyrics} is more than 70 times longer than that of D_{Title}. However, we desire a smaller weight of a term in the longer document. Therefore, besides the original intra-collection normalization (Formula 2.3), we must consider a new normalization on the average document lengths of local collections (inter-collection document length normalizations). One possible solution might be merging all the collections into a **global collection** and using a single global *avgdl*. For the above example, the global *avgdl* is 116 (see Table 1 for details), and the *ndl* values for D_{Title} and D_{Lyrics} become 0.81 and 1.21 respectively, which are more reasonable values. However, this solution can cause another problem due to the large diversity in *avgdl* values in different collections. In our example, the global *avgdl* (116) is dominated by the very long text column *Lyrics* (239). Normalizations on short columns are lost as shown in the following new example. Consider the two documents (i.e. two text column values) in tuples *a1* and *b1* (we call them D_{a1} and D_{b1}) as shown in Figure 1. Both D_{a1} and D_{b1} contain the keyword "D12"; the length of D_{a1} is 1 and the length of D_{b1} is 2. And their *ndl* values are 0.802[2] and 0.803 respectively. If "D12" occurs in another document whose length is 10, its *ndl* is 0.817. This example shows that this solution fails in the intra document length normalization in the following way: For two different text columns of very different average lengths, which are much shorter than the global *avgdl*, their *ndl* values differ very little. However, we desire significant difference between them.

We propose Formula 7 to consider both intra-collection and inter-collection document length normalization: (1) we still maintain a local collection for each text column and use Formula 2.3 for intra-collection normalizations, and (2) then we normalize the

$$ndl = \left((1-s) + s * \frac{dl}{avgdl}\right) * \left(1 + \ln(avgdl)\right) \qquad \textbf{(7)}$$

local *avgdl* for the local collection using 1+ln(*avgdl*). This formula can solve the problems shown in the above two examples. The new *ndl* values computed using Formula 7 for the keyword "how" in a song title document D_{Title} and a song lyrics document D_{Lyrics} (suppose their lengths are equal to their own *avgdls*) become 2.17 and 6.48 respectively[3]. Thus, the term weight in the long document becomes smaller. The new *ndl* values for the keyword "D12" in D_{a1} and D_{b1} (see Table 1 for their *avgdl* values) are 1.48[4] and 1.88 respectively. If "D12" occurs in a document whose length and *avgdl* are both 10, the new *ndl* value is 3.3. They are more reasonable normalizations. Note that this normalization is independent on any keyword in the query.

4.2.3 Document Frequency Normalization

Collections have different vocabularies, and term distributions are different. Document frequency normalization also has the problem of local *vs.* global collections. For example, in the collection of *Name* of *Artist*, terms that are people names have high document frequencies. However, document frequencies of these terms are usually low in the collection of *Lyrics* of *Song*. In addition, the total numbers of documents in *Name* of *Artist* is much smaller than that in *Lyrics* of *Song*. Thus, for a term that is a person's name, its *idf* value in *Name* of *Artist* is usually smaller than that in *Lyrics* of *Song*. If this term appears in 1/100 of the documents in *Name* of *Artist*, and 1/1000 of the documents in *Lyrics* of *Song*, then its *idf* values in the two collections are 4.6 and 6.9 respectively. Furthermore, even in a rather large relational database, some tables may have a small number of tuples, and document frequency statistics in columns of these tables become unreliable. Therefore, we propose to use global document frequency statistics. The idf^g as shown in Formula 8 remains the same as that in Formula 2.2 except we use global statistics, where

$$idf^{g} = \ln\frac{N^{g}}{df^{g}+1} \qquad \textbf{(8)}$$

[2] 0.802=(1-*s*)+*s**1/116, where *s* is 0.2, and 116 is the global *avgdl*. Computations for 0.803 and 0.817 are similar.

[3] 2.17=1 * (1+ln(3.21)), and 6.48= 1 * (1+ln(239))

df^g is the global document frequency of the term (number of documents in the whole database, i.e. all text column values, that the term occurs), and N^g is the total number of documents (i.e. the total number of text columns values) in the whole database.

4.2.4 *Inter-Document Weight Normalization*

With the above three normalizations, the term weight in a document D_i in T is computed as shown in Formula 5.1, where *ntf* (normalized term frequency) is computed by Formula 2.1, *idf* (inverted document frequency) is computed by Formula 8, *ndl* (normalized document length) is computed by Formula 7, and *Nsize(T)* (normalized tuple tree size) is computed by Formula 6. For *Comb()*, we can simply sum up the term weights in all of the documents in *T*. However, a term tends to appear more frequently in a *T* with a larger size. We use Formula 9 to normalize *weight(k, T)*, where max*Wgt* is the maximum *weight(k, D_i)* and *sumWgt* is the sum of *weight(k, D_i)* for all D_i in *T*. The idea behind Formula 9 is borrowed from term frequency normalization (Formula 2.1): We consider the max*Wgt* as one unit of term frequency and *sumWgt/* max*Wgt* as the total term frequency.

$$Comb() = \max Wgt * \left(1 + \ln\left(1 + \ln \frac{sumWgt}{\max Wgt}\right)\right) \qquad \textbf{(9)}$$

4.3 Schema Terms in Query

Another new problem for keyword search in relational databases is that user queries usually contain two types of terms. The first type is terms that are matched with text column values, and we call them **value terms**. The second type is terms that are matched with the names of text columns, tables and databases, and we call them **schema terms**. For example, the keyword "lyrics" in Query 2 and the keyword "album" in Query 3 are both schema terms. Schema terms usually do not occur in text values; thus the weight of such a schema term in relevant tuple tress is usually 0. However, if a text value happens to contain some schema terms, the weights of such schema terms are non-zeroes. One example is a query "lusher the singer's lyrics to burn". There happens to be a song titled "The Singer", and it is ranked top 1 for the query. Obviously, this is not what the user wants. The relevant one should be the song titled "Burn" by the artist "Usher". Although the keyword "singer" matches the name of the attribute *Artist*, the text column values in this answer do not contain "singer". The weight of "singer" in this answer is 0, which causes the problem of the relevant answer having a lower similarity value. We use a simple method to solve this problem. For each text column, each table and the database, we identify a set of synonyms that are their names. For example, we generate {"artist", "band", "singer"} for the table *Artist*, {"album"} for the table *Album*, {"song"} for the table *Song*, {"lyrics"} for the column *Lyrics*, etc. For these schema terms, besides their global document frequency values based on their occurrences in text column values, we also assign a **schema-based document frequency** value for each of them: for a schema term *k*, if it is a synonym of a text column or a synonym of table containing one text column, we assign the largest document frequency value among all terms in the local collection that corresponds to the text column to *k*, and if it is a synonym of a table containing more than one text column, we assign the largest document frequency value in the collection that corresponds to a union of all the text columns in the table to *k*. Thus, if *k* is a schema term in a given query, *k* does not occur in a tuple tree *T* (i.e., its term frequency is 0) but *k* corresponds to the name of a table or a column involved in *T*, then *weight(k, T)* computed by Formula 5.2 is 0. However, we now assume that the term frequency of a schema term *k* in each of its corresponding text columns involved in *T* be 1, and its df^g be its schema-based document frequency. Thus, we can get a *weight(k, T)>0.* If *k* is a schema term, *k* occurs in D_i of *T*, and *k* corresponds to the text column of D_i or the table containing D_i, then we compute two values of *weight(k, D_i)*, one is computed by considering *k* as a value term and the other is computed by considering *k* as a schema term. Then we choose the larger weight as the final *weight(k, D_i)* because we assume that each term in the query has only one meaning. For the above example, the final similarity value between the query and the first answer remains unchanged, however the similarity value between the query and the second answer is increased because the term "singer" has a non-zero weight and the second answer becomes top 1 now. The identification of schema terms needs manual work or can be done semi-automatically using WordNet [25]. However, the workload is rather trivial because (1) the number of schema terms is small (only 6 schema terms {"artist", "band", "singer", "album", "song", "lyrics"} are used by users in lyrics search), (2) a well designed database should have information (i.e. schema descriptions) that helps generate these schema terms.

4.4 Phrase-based Ranking

Research in IR [14] has shown that phrase-based search can improve effectiveness. Google [15] also utilizes proximity information. In relational databases, many text columns are name entities and favor proximity and phrase search. Consider the sub-query "how come" in Query 2. It is a title of a song and should be considered to be a phrase. Another example is "Code Red", a name of an artist (band). Although both keywords are common words, the phrases have specific meanings. To identify phrases from a query, Liu et al. [14] use natural language software and a dictionary to analyze the query. Instead, we only utilize term position information in documents (term position information is stored in the inverted indexes), without using any additional source. Whether a sub-query of a query is a phrase is tuple tree dependent. If a sub-query of *Q*, $P=\{k_i,k_{i+1},..k_j\}$, where $i<j$, appears in a document *D*, and k_{i-1} does not appear in an adjacent location to k_i in this occurrence of *P* in *D*, and k_{j+1} does not appear in an adjacent location to k_j in this occurrence of *P* in *D*, then we define it as an occurrence of the **phrase** *P* in *D*. We modify Formula 5.1 into Formula 10 to compute *weight(P,D)***,** the weight of a phrase *P* in a document *D*. Since the tuple tree size and document length normalizations are independent to any query keyword, the denominator remains unchanged. The numerator is replaced by *npf*idf*. For the *idf* component, we assume that *idf* of a phrase is the sum of the *idfs* of all the individual keywords in the phrase for computational simplicity (Formula 10.2). For *npf* (normalized phrase frequency), the same term frequency normalization function (Formula 10.1) is used for a phrase (*pf* is phrase frequency in D_i); in addition, we add a new normalization factor to boost the adjacency of the words in a phrase: *1+ln(length(P))*, where *length(P)* is the number of keywords in *P*. For example, a phrase with two words is 1.7 times more important than the sum of individual term weights when they do not form a phrase.

4 1.48=((1-*s*)+*s** (1/1.89))*(1+ln(1.89)), where 1.89 is *avgdl* of *Artist.Name* and s is 0.2. Computations for 1.88 and 3.3 are similar.

$$weight(P, D_i) = \frac{npf * idf}{ndl * Nsize(T)} \quad \textbf{(10)}$$

$$npf = (1 + \ln(1 + \ln(pf))) * (1 + \ln(length(P))) \quad \textbf{(10.1)}$$

$$idf = \sum_{k \in P} \ln(\frac{N^g}{df^g{}_k + 1}) \quad \textbf{(10.2)}$$

The definition of an occurrence of P in D causes a new problem, which is illustrated by the following example. Suppose $Q=\{1, 2, 3, 4\}$ and a document D in T is $\{.. 1, 2, 3 .. 2, 3, 4 .. 2, 3, 4 .. 1, 2 .. 1 ..\}$. By the definition, we have phrases $\{1, 2, 3\}$, $\{2, 3, 4\}$ and $\{1, 2\}$ in D, and their *pfs* are 1, 2 and 1 respectively. However, $\{1, 2, 3\}$ and $\{2, 3, 4\}$ overlap and neither of them contains the other one. We assume that each occurrence of a phrase or of a keyword in a given query have a unique meaning in a tuple tree. Thus, we only allow one of $\{1, 2, 3\}$ and $\{2, 3, 4\}$ to be a phrase for a tuple tree. We propose an algorithm (Figure 7) to re-identify phrases in a document. For example, if the phrase weight of $\{1, 2, 3\}$ is higher, the output of the algorithm is $\{1, 2, 3\}$, $\{1, 2\}$ and $\{2, 3\}$. Their *pfs* are 1, 1 and 2 respectively. If the phrase weight of $\{2, 3, 4\}$ is higher, the output is $\{2, 3, 4\}$ and $\{2, 3\}$. Their *pfs* are 2 and 1 respectively. Then we use Formula 10 for phrases and Formula 5.1 for individual keywords to compute the final *weight(Q,T)*.

A phrase may contain shorter phrases or individual keywords, and their weights are computed independently. If we sum them up, we may have a problem that is similar to the inter-document weight normalization problem (Section 4.2.4). For phrases, we slightly modify Formula 9 to borrow the idea of the inter-document weight normalization. When we match a query against documents, we should always do maximal matching, i.e. maximal phrases. For a given query Q and an answer T, we first identify the set of phrases and keywords in T that are not contained in any longer phrases. We call this set a **concept set** (C^Q) in Q against T. Then for each concept c (either a phrase or a keyword) in C^Q, max*Wgt(c)* is computed as the maximum *weight(c, D_i)* over all D_i in T, and *sumWgt(c)* is the sum of all *weight(p', D_i)*, where *p'* is *c*, or any shorter phrase or keyword contained in *c* and D_i is a document in T. Suppose c has the maximum weight in D_i, then we (1) use *maxWgt(c)* as the concept weight of c in T, and then (2) bind c to the column of D_i, (the semantics of the concept c is bound to the text column of D_i). To avoid assigning a column to a single word concept that has a high document frequency and appears in a long document, we set a threshold *th*. If max*Wgt(c)<th*, we do not assign the column to *c*. For example, the keyword "by" in Query 2 has a high *df* and occurs in a long document (a *Lyrics* column value in tuple *s1*), and it has a small max*Wgt* value, thus we do not assign *Song.Lyrics* to it. However, we assign *Artist.Name* to the concept "D12" and "*Song.Title*" to the concept "how come". Thus, a **phrase model** can return answers with semantics (i.e. the correspondences between concepts in the queries and the columns in the database).

For a query with multiple terms (such as Query 3), a document that contains only some highly weighted terms (due to high *idf* and/or high term frequencies even we have done *tf* normalization and/or short document length) may be ranked higher than a document that contains all moderately weighted terms. We want to rank the latter document higher. Our strategy is to use **Concept** ranking model, which is based on *Phrase* model. For each concept c_i in C^Q (described in the above paragraph), we compute a

```
Input: query $Q$, tuple tree $T=\{D_1, D_2, ..D_n\}$
Output: phrase occurrence set $C$
1. Use the definition to identify all phrases $\{P_1,..P_n\}$ in $T$. Put them into $C0$.
2. While $C0$ is not empty {
    2.1 Move the set of longest phrases from $C0$ to a new set $L$.
    2.2 While L is not empty {
        If there is no overlapped phrases in $L$ {
            Add L to $C$. Move each phrase that is contained in any phrase in $L$ from $C0$ to $C$.
        }
        Else {
            2.2.1 Choose $P_i$, $D_j$ such that $weight(P_i,D_j)$ has the maximum weight in $L$. Move $P_i$ from $L$ to $C$.
            2.2.2 For each $P_k$ in $L$ or $C0$ that overlaps with $P_i$ {
                2.2.2.1 Move $P_k$ out of $L$ or $C0$.
                2.2.2.2 Break $P_k$ into two parts. One is a sub-phrase or a keyword contained in $P_i$, and put it into $C$ if it is a phrase. The other one is the remaining part; and put it into $C0$ if it is a phrase.
                2.2.2.3 Move all phrases contained in $P_k$ from $C0$ to $C$.
}   }   }   }
3. Return $C$.
```

Figure 7: Phrase Identification Algorithm

weight(c_i, D_j) using Formula 5.1 if c_i is a keyword, and using Formula 10 if c_i is a phrase. The only exception is that we drop the *1+ln(1+ln(tf))* component from both formulas because we only take into consideration whether a concept appears but not its term frequency according to [14]. Then we choose the maximum weight(c_i, D_j) as weight(c_i, T), and sum up all these concept weights to obtain a **concept similarity** value *Sim(C^Q, T)*. The similarity between a tuple tree T and a query Q becomes a tuple *(Sim(C^Q,T), Sim(Q,T))*. The first component is the concept similarity and the second component is the term similarity computed by Formula 5.1 and Formula 9 respectively. For two tuple trees T_1 and T_2 and a given query Q, if (1) $Sim(C^Q, T_1) > sim(C^Q,T_2)$, or (2) $Sim(C^Q,T_1)=Sim(C^Q,T_2)$ and $Sim(Q,T_1) > Sim(Q, T_2)$, then T_1 is ranked higher than T_2.

5. EXPERIMENTAL RESULTS

Section 5.1 describes the data set. Section 5.2 introduces evaluation measures and the setups for comparison. Section 5.3 reports and discusses experimental results on search effectiveness.

5.1 Data Set

Database: We use a lyrics database in our experiments. It is reported in [20] that lyrics search is one of the most popular search in various major search engines. We crawled an entire lyrics web site in Aug. 2005, and converted the data into a relational database whose schema is shown in Figure 3. There are more than 177K songs in the database. Table 1 gives basic statistics of the database. We do not stem words and do not use stop word except "the".

Query Set: We use a set of 50 queries for evaluation. These queries are obtained from a subset of a one-week period user query log on a commercial search engine. By using IP addresses and short durations between the times when queries are submitted, we detect that a sequence of queries are submitted by the same users, and each such query is a small variant of the next query in the sequence. By analyzing such queries, the intentions of the users can be found out. This allows us to make relevance assessments of the retrieved results relative to the queries. We randomly choose 50 lyrics queries from the query log (see Appendix for the 50 queries). For each query, we identify a set of

Table 1: Lyrics Database Statistics

Column	# of doc	*avgdl*
Artist.Name	3,691	1.89
Album.Title	15,160	2.67
Song.Title	177,231	3.21
Song.Lyrics	177,231	239

relevant answers (tuple trees) in the crawled lyrics database using pooled relevance judgment: we do retrieval using many different methods including Google; for each method, we obtain the top 10 answers and then we get the union of the sets of the top 10 answers; then we judge the relevance for each of the answers in the union. This way of relevant judgment is used in TREC [9]. There are totally 135 relevant answers (2.7 per query on the average). The maximum number of relevant documents for a query is 14, and the minimum number is 1. The average length of a query is 6.7; the maximum length is 20 and the minimum length is 2. The average length of such queries is substantially larger than that of typical Internet queries, but in order to specify artists, songs and lyrics, the longer lengths are needed. We further classify them into two sets: simple queries (with only one non-schema concept) and complex queries (with two or more non-schema concepts, and they are underlined in Appendix). Each set contains 25 queries. We expect that our ranking strategy work better on complex queries than on simple queries. We also note that all our queries contain schema terms.

5.2 Measures and Comparison Setup

We use 4 measures to evaluate the different aspects of search effectiveness. (1) Number of top-1 answers that are relevant. (2) Reciprocal rank. They measure how good the system is to return one relevant answer. (3) 11-point precision/recall and (4) MAP (see Section 3.1 for Measures 2, 3 and 4). They measure the overall effectiveness for *top10* answers in our experiments (other *topk* numbers have very similar results). Result comparison serves two purposes: investigating how different factors in our ranking strategy affect search effectiveness, and comparing our ranking strategy with related work to demonstrate its superiority.

The comparison is conducted on three dimensions. (1) Whether the system identifies schema terms (Section 4.3). (2) The four normalization factors described in Section 4.2. (3) The four search models. The first model (*All-Word*) assumes AND semantics (i.e. an answer must contain all keywords) and ranks answers by number of tuples in the tree in ascending order. The second one (*Keyword*) assumes OR semantics and uses IR style ranking. But it only considers individual keywords in queries. The third one (*Phrase*) also considers phrases in queries. The last one (*Concept*) is a *Phrase* model that uses concept-based ranking.

5.3 Results and Discussion on Effectiveness

5.3.1 Identification of Schema Terms

Table 2 gives the reciprocal rank values for all the four models (with all four normalizations) with (*w/s*) and without (*w/o*) identification of schema terms. The percentage of improvement (*imp*) of the run with this factor over the run without this factor is also reported. These values are also reported for complex and simple queries respectively. The results show that identifying schema terms is extremely important for *All-Word* search model, because it assumes an AND semantics and schema terms are not usually contained in values. This factor seems to have marginal effect on other models (only slight improvement for *Concept* model on complex queries and *Keyword* model for all) because in Lyrics database, occurrences of schema terms in values are rare.

Table 2: Identifying Schema Terms

Model	All			Complex			Simple		
	w/s	w/o	imp	w/s	w/o	imp	w/s	w/o	imp
Concept	0.871	0.850	2.5%	0.900	0.856	5.1%	0.833	0.844	-1.3%
Phrase	0.790	0.794	-0.5%	0.828	0.828	0.0%	0.752	0.759	-1%
Keyword	0.696	0.662	5.1%	0.799	0.763	4.7%	0.594	0.560	6.1%
All-Word	0.245	0.020	-	0.316	0.000	-	0.158	0.040	-

Table 3: Tree Size Normalization

Model	All			Complex			Simple		
	w/s	w/o	imp	w/s	w/o	imp	w/s	w/o	imp
Concept	0.871	0.511	70.5%	0.900	0.178	406%	0.833	0.844	-1.3%
Phrase	0.790	0.444	77.9%	0.828	0.129	536%	0.752	0.759	-1%
Keyword	0.696	0.357	93.3%	0.799	0.124	537%	0.594	0.589	0.8%

Table 4: Document Length Normalizations

Model	All			Complex			Simple		
	w/s	w/o	imp	w/s	w/o	imp	w/s	w/o	imp
Concept	0.871	0.488	78.3%	0.900	0.395	228%	0.833	0.581	42.9%
Phrase	0.790	0.585	35.0%	0.828	0.454	82.4%	0.752	0.715	4.9%
Keyword	0.696	0.492	40.2%	0.799	0.477	33.0%	0.594	0.507	17.2%

Table 5: Document Frequency Normalization

Model	All			Complex			Simple		
	w/s	w/o	imp	w/s	w/o	imp	w/s	w/o	imp
Concept	0.871	0.708	23.0%	0.900	0.807	11.5%	0.833	0.600	38.8%
Phrase	0.790	0.676	16.9%	0.828	0.779	6.3%	0.752	0.573	31.2%
Keyword	0.696	0.685	1.6%	0.799	0.824	-3.0%	0.594	0.548	8.4%

Table 6: Inter-Document Weight Normalization

Model	All			Complex			Simple		
	w/s	w/o	imp	w/s	w/o	imp	w/s	w/o	imp
Concept	0.871	0.871	0.0%	0.900	0.898	0.2%	0.833	0.844	-1.3%
Phrase	0.790	0.680	16.2%	0.828	0.756	9.5%	0.752	0.604	24.5%
Keyword	0.696	0.591	17.8%	0.799	0.685	16.6%	0.594	0.497	19.5%

Table 7: Combining All Normalizations

Model	All			Complex			Simple		
	w/s	w/o	imp	w/s	w/o	imp	w/s	w/o	imp
Concept	0.871	0.358	143.3%	0.900	0.068	1224%	0.833	0.648	28.5%
Phrase	0.790	0.351	125.1%	0.828	0.076	989%	0.752	0.627	19.9%
Keyword	0.696	0.248	180.6%	0.799	0.060	1232%	0.594	0.438	35.6%

Schema terms rarely form phrases, so it has no effect on *Phrase* model. In the following sections, all runs use schema terms identification if not otherwise specified.

5.3.2 Four New Normalizations

We give four tables (Tables 3-6) to show how each new normalization factor improves keyword search effectiveness using three search models (*Keyword, Phrase,* and *Concept*). Table 7 shows how all the new factors improve the overall effectiveness.

Table 3 gives reciprocal rank values for all the three models (with all other three normalizations) with (*w/s*) tree size normalization and without (*w/o*) it (but using the simple normalization in Formula 4). The results show that this factor has tremendously significant effect for all three models on complex queries, because the relevant answers ($Size(T) > 1$) are normalized too much even though we use $(Size(T)+1)/2$ instead of $Size(T)$ [11] in the experiments. It is also not surprising that it has little affect on simple queries, whose answers contain only single tuples and should be ranked higher than all multi-tuple trees.

Table 4 gives reciprocal rank values for all the last three models (with all other three normalizations) with (*w/s*) our document length normalization and without (*w/o*) it. The results show that this factor is also very critical for all three models on all queries.

This factor affects the complex queries more than simple queries because it deals with diverse types (in terms of the document length) of columns. This factor also affects *Concept* and *Phrase* models more than *Keyword* model for complex queries because the first two models boast the weights of long phrases, and longer documents tend to contain more and longer phrases.

Table 5 gives the reciprocal rank values for all the last three models (with all other three normalizations) with (*w/s*) our document frequency normalization and without (*w/o*) it. It significantly affects *Concept* and *Phrase* models on all queries. It has little effect on Keyword model when all other three normalizations have been applied. The results also show that this factor affects simple queries more than complex queries.

Table 6 gives the reciprocal rank values for all the last three models (with all other three normalizations) with (*w/s*) our inter-document weight normalization and without (*w/o*) it. The results show that this factor significantly affects *Phrase* and *Keyword* models but less significantly than the tree size and the document length factor. It does not improve effectiveness on *Concept* model because this model ranks answers with concept similarity in the first order and this factor has no effect on the concept similarity (we always use max*Wgt*).

Finally, we use Table 7 to demonstrate the overall improvement by comparing models with all the four normalization factors to the same models without any of them. Without any of the four normalizations, all models perform very poorly on complex queries. These normalizations also improve simple queries, but less significantly. We conclude that (1) all these four normalization factors are critical to search effectiveness, (2) the first two factors improve search effectiveness more significantly than the last two, (3) and these normalization affect complex queries more than simple queries.

5.3.3 Four Models and Related Work

In Figure 8 and Table 8, *Concept, Phrase* and *Keyword* are models that use identification of schema terms with all four normalizations and with OR semantics. Figure 8 shows the 11-point precision/recall graph for the four models with all normalizations and with identification of schema terms. MAP values are also given (following the labels). Table 8 also gives results of related work. *Concept2* is the model that does not identify schema terms. *Related1* is the *All-Word* model with identification of schema terms; it is used as an upper bound for the method used in DBXplorer [1] and DISCOVER [10]. *Related2* is the *Keyword* model with identification of schema terms and without any of the four new normalizations; it is used as an upper bound for the method used in Hristidis et al. [11] with OR semantics. We also report results of *Keyworda* (i.e. the *Keyword* model with AND semantics) and *Related2a* (i.e. the *Related2* model with AND semantics, which is used as an upper bound for the method in Hristidis et al. [11] with AND semantics). We also compare our results against that obtained by Google. We note again that Google retrieves from text databases instead of relational databases. To evaluate search effectiveness of Google, we submitted each query to Google. Among its top 10 results, we identified the first relevant web page as follows. (1) Based on our relevant assessment, if the relevant answer is a tuple tree with a single tuple, and if a Google result is a web page with the same information as the tuple, then it is relevant. (2) If the relevant answer is a tuple tree with both an *Artist* tuple and an *Album* tuple, and if a Google result is a web page with the same information as the *Album* tuple (this web page usually contains *Artist* information), then it is relevant. (3) If the relevant answer is a tuple tree with a *Song* tuple and one or more *Artist* and/or *Album* tuples, and if a Google result is a web page with the same information as the *Song* tuple (this web page usually contains both *Album* and *Artist* information), then it is relevant. We do not use the MAP measure in Table 8 due to two reasons: (1) it is highly correlated with the reciprocal rank measure (R-Rank in Table 8) in our experiments, and (2) Google has many copies of lyrics databases from different web sites, and the same answers are often returned multiple times. We also report the number of top-1 answers that are relevant (#Rel in Table 8).

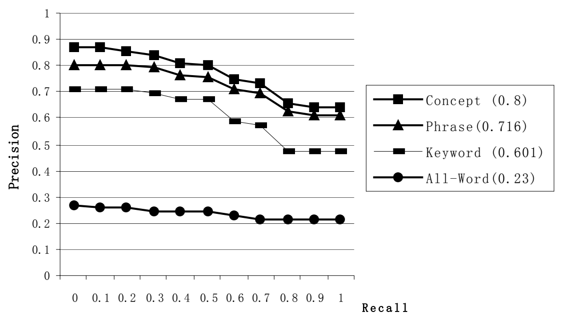


Figure 8: 11-pt Precision/Recall and MAPs for 4 Models

Table 8: Model and Related Work Comparison

Model	All		Complex		Simple	
	# Rel	R-Rank	# Rel	R-Rank	# Rel	R-Rank
Concept	39	0.871	20	0.900	19	0.833
Concept2	40	0.850	20	0.856	20	0.844
Phrase	35	0.790	18	0.828	17	0.752
Keyword	32	0.696	18	0.799	14	0.594
Keyworda	29	0.615	13	0.552	16	0.678
[Related1]	8	0.245	6	0.316	2	0.158
[Related2]	10	0.248	1	0.060	9	0.438
[Related2a]	21	0.491	9	0.428	12	0.554
[Google]	*34*	*0.749*	*15*	*0.709*	*19*	*0.788*

The results show that *Related1* [1, 10] and *Related2* [11] are not as effective as our method. The comparison between *Related2* and *Related2a*[5] shows that, for the ranking method used in Hristidis et al. [11] (Formula 4 in this paper), AND semantics yields significant improvement over OR semantics, especially for complex queries (0.428 vs 0.06). The reason is that AND semantics tends to exclude tuple trees containing only one tuple (the ranking scores of these tuple trees using OR semantics are high due to their small size) which contains only a subset of the query keywords (thus they are less relevant). For example, for *Related2* (OR semantics) all three relevant answers for the 26th query are ranked below the top 10 results; all the top 10 answers are tuple trees that only contain individual tuples and a subset of the query keywords. For *Related2a* (AND semantics), most of the top 10 results in *Related2* are excluded because they do not contain all keywords, and among the top 10 results, two are relevant answers that contain multiple tuples. The comparison between *Keyword* and *Keyworda* shows that, for our ranking method (Formulas 5.1 and 5.2), OR semantics yields significant

[5] For both *Related2* and *Related2a*, the results of not using schema terms are less effective than those of using schema terms, so they are not reported.

improvement over AND semantics. The comparison between *Keyword* and *Related2,* and that between *Keyworda* and *Related2a* show that our ranking method yields significant improvement for both OR semantics (181%) and AND semantics (25.3%) over the ranking method in Hristidis et al. [11]. *Concept* and *Phrase* models outperform Google on all queries. Our best result (*Concept2*) has a 16.3% improvement over Google and 77.4% over the best (*Related2a*) of Hristidis et al. [11]. Even when we do not consider schema terms (without any manual work involved), the improvement is 13.5%. From the point view of IR effectiveness, these improvements are significant. For complex queries whose answers are assembled by joining tuples from different tables, all three models outperform Google (*26.9%* for *Concept*, *20.7%* for *Concept2*, *16.8%* for *Phrase*, and *12.7%* for *Keyword*) more significantly than for simple queries. And the improvement over Google on simple queries is less significant. We conclude that (1) in terms of search effectiveness, *Concept* > *Phrase* > *Keyword* > *All-Word*, and (2) our ranking strategy outperforms related work in keyword search in relational databases very significantly and outperforms the state-of-the-art IR method (Google) significantly.

6. RELATED WORK

Keyword search in relational databases [1, 2, 10, 11] has recently emerged as a new research topic. DBXplorer [1], DISCOVER [10], Hristidis et al. [11] and BANKS [2] are systems that support free-form keyword search on relational databases. They return tuple trees as answers for a given keyword query. One focus of the above works is to generate tuple trees efficiently. DBXplorer, DISCOVER and Hristidis et al. [1, 10, 11] construct a set of join expressions (called answer graph in our paper) for a given query, and then evaluate these join expressions to produce tuple trees. BANKS [2] finds all tuple trees from the data graph directly using a Steiner tree algorithm. In the data graph, they use PageRank style methods to assign weights to tuples and assign weights to edges between tuples. DBXplorer [1], DISCOVER [10] and BANKS [2] assume AND semantics for an answer. To rank answers, they either [1,10] simply use the number of joins in the tuple trees or [2] use a combination of tuple weights and edge weights in a tuple tree without any IR-style ranking method. Hristidis et al. [11] assume OR semantics for answers. To rank answers, they incorporate the IR relevance ranking in a straightforward manner. Many new factors that are critical to search effectiveness are not investigated. Goldman et al. [7] propose a very simple query language with two sets of keywords: *Find* set and *Near* set. Two result sets of database objects are obtained for the two sets of keywords. Then the *Find* result set is re-ranked using distance information between the two sets. Luo et al. [9] combine non-text data values with keyword search on text columns. They concentrate on the efficiency issue of integrating inverted indexes with non-text values. Our work differs from all the above works in three aspects. (1) All of the above works focus on the efficiency issue, not on search effectiveness, which is as important as the efficiency in keyword search, if it is not more important. The only work that mentions effectiveness is BANKS, but they only use 6 queries without any standard evaluation. ObjectRank [3] uses an authority-based ranking strategy for keyword search in relational database. They return individual tuples instead of tuple trees as answers. The ranking score for a tuple is a combination of its ObjectRank value and term frequencies of the query keywords. On effectiveness, they only report non-standard effectiveness measures with two queries. In contrast, we conduct comprehensive experiments using a real database and real user queries with standard evaluation results. (2) Among the above related works, they either do not use IR style ranking at all, or they use it without sufficient consideration of effectiveness (e.g. [11]). In contrast, we propose a novel IR style ranking strategy for the new problem, which is very effective. (3) None returns answers with semantics as we do.

Another different but related research topic is keyword search in XML databases [6, 8, 13]. Florescu et al. [6] extend an XML query language with keyword search. They do not use IR style ranking. XRANK [8] returns XML document fragments as answers. To rank answers, they combine the granularity, hyperlink and keyword proximity information with simple IR style ranking. Both Florescu et al. [6] and XRANK [8] require users to know the XML schema and use a structured query language. Li et al. [13] propose a schema-free query language for XML, but it is still not structure-free. Kaushik et al. [12] propose an approach to integrating inverted indexes with structured indexes to support more efficient keyword search in XML. Besides the difference between relational and XML databases, none of the above work is for free-form keyword search, and they either incorporate IR-style ranking in a straightforward manner without considering the critical factors in this paper or do not incorporate IR ranking at all. XSEarch [5] proposes a free form keyword query language on XML. To rank answers, which are trees of XML nodes, they combine a simple *tf*idf* IR ranking with the size of the tree and the node relationship information. This work only gives examples (but without experimental results) on search effectiveness.

Keyword search has been extensively studied in IR [17, 18, 27] and web search [4, 21]. Sacks-Davis et al. [16] tackle a similar problem with ours from IR's point of view. They investigate indexing methods to support structures in documents. But they do not propose new ranking strategy for the new problem. Recently, link structures [15] have been successfully incorporated with IR ranking in web search. [2, 3, 7, 8] use link structures in the relational or XML databases. The usage of link structures is orthogonal to the usage of IR ranking. We will incorporate link structures in keyword search in our future work.

Major RDBMSs [22, 23, 24] have incorporated IR ranking strategies into their full-text search. However, attribute names must be specified for keywords in SQL queries; free form keyword search is not supported. Our work can be applied into the core of a RDBMS to support free form keyword search (without specifying attribute names). Many web search engines and enterprise search engines are built on structured databases. For example, Google [21] provides product search (froogle), academic paper search (Google scholar), and etc. Amazon.com provides book and product search. Dealtime.com provides product comparison service. IMDB.com has a form-based query interface to search movie information. Although technical details of how they process queries with keywords are absent, experience implies that they do not fully utilize the combination of text data and the database structures. Our work can be applied into these applications to improve search effectiveness.

7. CONCLUSION AND FUTURE WORK

Keyword search allows non-expert users to find text information in relational databases with much more flexibilities. In this paper, we proposed a novel ranking strategy for **effective** keyword search in relational databases. A given keyword query is processed in three steps. (1) The system generates all answers (tuple trees) for the query. (2) The system computes a ranking score for each answer and ranks them. (3) Finally, *topk* answers are returned with semantics.

Our ranking strategy is novel. It identifies and uses four new normalization factors that are critical to search effectiveness: (1) tuple tree size normalization, (2) document length normalization, (3) document frequency normalization and (4) inter-document weight normalization. Schema terms are identified and are processed differently from value terms. Our strategy also uses phrase-based and concept-based models to improve search effectiveness further. And the concept-based model can also return answers with semantics. Comprehensive experiments were conducted using a real world lyrics database and a set of queries collected by a major search engine. Standard evaluation results were reported. The results show that: (1) all the four new normalization factors are critical to search effectiveness (the first two factors improve effectiveness more significantly than the last two, and they improve effectiveness more significantly on complex queries than on simple queries); (2) phrase-based search and concept-based search improve effectiveness significantly; (3) our strategy is significantly better than related works and significantly outperforms Google. Our approach not only can be used at the application level for keyword search in relational databases, but also can be incorporated into the core of a RDBMS.

We plan to utilize link structures (primary key to foreign key relationships as well as some hidden join conditions), and some non-text columns (for example, user review rates on a product, box-office income of a movie, and whether an actor won an Oscar award) in the relational databases along with pure text data. By combining these three pieces of information, we hope to improve search effectiveness further. Furthermore, we plan to investigate the efficiency issue. Finally, we plan to conduct experiments with more real world databases (for example, movie databases, academic paper databases, product databases and job databases) and more user queries.

8. ACKNOWLEDGEMENTS

This work is supported in part by NSF grants: IIS-0208434 and IIS-0208574. We thank the anonymous reviewers and Prof. Yannis Papakonstantinou for helpful comments. We thank AOL for providing web search query logs.

9. REFERENCES

[1] S Agrawal, S Chaudhuri, G Das: DBXplorer: A system for keyword-based search over relational databases. ICDE 2002

[2] G. Bhalotia, A. Hulgeri, C. Nakhey, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. ICDE 2002

[3] A. Balmin, V. Hristidis,Y. Papakonstantinou: Authority Based Keyword Queries in Databases using ObjectRank. VLDB 2004

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. WWW 1998

[5] S. Cohen, Jonathan Mamou, Yaron Kanza, Yehoshua Sagiv: XSEarch: A Semantic Search Engine for XML. VLDB 2003

[6] D. Florescu, I. Manolescu, and D. Kossmann. Integrating keyword search into XML query processing. WWW 2000.

[7] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity Search in Databases. VLDB 1998.

[8] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. SIGMOD 2003.

[9] L. Guo,J. Shanmugasundaram, K. Beyer, E. Shekita:Efficient Inverted Lists and Query Algorithms for Structured Value Ranking in Update-Intensive Relational Databases. ICDE 2005

[10] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. VLDB 2002.

[11] V. Hristidis,L. Gravano,Y. Papakonstantinou:Efficient IR-Style Keyword Search over Relational Databases.VLDB 2003

[12] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. SIGMOD 2004

[13] Y. Li, Cong Yu, H. V. Jagadish: Schema-Free XQuery. VLDB 2004

[14] S. Liu, F. Liu, C. T. Yu, Weiyi Meng: An effective approach to document retrieval via utilizing WordNet and recognizing phrases. SIGIR 2004.

[15] L. Page, S. Brin, R. Motwani and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web, Technical Report, 1998

[16] R. Sacks-Davis, Tuong Dao, James A. Thom, Justin Zobel Indexing documents for queries on structure, content and attributes. ISDM 1997

[17] A. Singhal, Chris Buckley, Mandar Mitra: Pivoted Document Length Normalization. SIGIR 1996

[18] A. Singhal. Modern information retrieval: A brief overview. IEEE Data Eng. Bull. 24(4), 2001

[19] E. M. Voorhees. Overview of the TREC-9 Question Answering Track. TREC 2000

[20] Pew Internet & American Life Project Report: Search Engine Users, 2005. www.pewinternet.org/pdfs/PIP_Searchengine_users.pd

[21] Google. www.google.com/ 2005

[22] DB2 Text Information Extender. 2005 http://www.ibm.com/software/data/db2/extenders/textinformation/index.html

[23] Micorsoft SQL Server 2000. www.microsoft.com/sql/ 2005

[24] MySQL. dev.mysql.com/doc/mysql/en/Fulltext_Search.html.

[25] G. A. Miller. WordNet: A lexical database for English. CACM, 38(11):39--41, 1995.

[26] G. Salton and M. McGill. Introduction to Modern Information Retrieval. McGraw-Hill, 1983

[27] D. Grossman and O. Frieder, Information Retrieval: Algorithms and Heuristics, Springer Publishers, 2nd Edition 2004

Appendix: 50 Queries (Complex and Simple)

1 to me lyrics by lionel richie   2 inner smile texas lyrics
3 lionel richie lyrics   4 lionel richie lyrics you mean more to me
5 avril lavigne lyrics for the album under this skin
6 avril lavigne lyrics   7 lyrics to and all i can taste is this moment
8 when i said i do lyrics   9 when i say i do lyrics
10 when i say i do lyrics clint black
11 clint black and wife when i say i do lyrics
12 Hanson i don't know lyrics   13 lyrics woman's worth
14 lyrics maxwell woman's worth   15 lyrics maxwell a woman's work
16 lyrics for how come by D12   17 usher the singer's lyrics to burn
18 lyrics that go "you ran me off the road..you're no longer laughing..i am not drowning fast enough"   19 ashlee simpson lyrics
20 lyrics to the songs on simpson's debut album "autobiography"
21 lyrics to rascal flatts moving on   22 lyrics to hanson wheres the love
23 christina millian-dip it low lyrics   24 christina milian-dip it low lyrics
25 Lil Jon "Get Low Remix" lyrics   26 jojo-leave lyrics
27 lyrics to ribbon in the sky   28 slow motion lyrics
29 i like that lyrics   30 the way i am lyrics
31 i'm just me and thats all i can be lyrics
32 this yo song ma lyrics   33 get no better lyrics
34 the games have all changed since i been around lyrics
35 cassidy lyrics
36 though i have to find all the answers to my question lyrics
37 run right through me lyrics   38 never ever have felt so low lyrics
39 i keep searching lyrics   40 have you ever felt so low lyrics
41 all the vocabulary runs through my head lyrics
42 lyrics to shania twains song my heart only breaks when its beating
43 lyrics to ashlee simpsons song pieces of me
44 lyrics to amazed by lonestar   45 lyrics to I believe by Fantasia
46 lyrics to soundtrack to Cradle to the Grave
47 lyrics to focus on the Cradle to the Grave
48 lyrics to talk about our love by brandy
49 edwin mccain lyrics   50 edwin mccain lyrics better when I'm older